# Tree-Based Buffer Management in Real-Time Database Systems

Uwe Brinkschulte
*Institute for Microcomputers and Automation, University of Karlsruhe*
*Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany, E-mail: brinks@ira.uka.de*

**Abstract**

*This paper deals with buffer managament for tree-based access-path structures in real-time database system kernels. A predictable and efficient buffer replacement strategy for B- and B+-trees is introduced. The strategy called **LC** (Level Control) uses the structural information known about the trees and the tree search to determine a node to be replaced. The basic idea is to use the level of a node as replacement criteria. This allows a certain prediction of buffer hits and misses for tree search operations, even if the item's value searched for is unknown. A small prediction interval can be calculated for search operations and a larger interval for update operations. Simulations show the correctness of these calculations and the efficiency of LC for B- and B+-trees compared to common other replacement strategies.*

## 1. Introduction

The use of a main-memory buffer is a good way to increase the speed of systems dealing with mass storage such as database systems. Therefore, most real-time database systems try to keep the whole database (or at least the whole actual working set) resident in the buffer at runtime. This avoids slow mass storage access and makes the system behavior more predictable. If the database (or working set) becomes too big to entirely fit in the buffer, a replacement strategy is needed. A good replacement strategy keeps the average system performance still high, but it becomes very difficult to predict buffer misses and the resulting performance for a single database operation (e.g. search).

In the approach presented here, a partial solution for this problem is proposed. This solution works for a specific part and type of data structures kept in the database files: the access-paths based on B or B+-trees. The data to keep in the buffer for these structures are the tree nodes. A buffer replacement strategy for these tree nodes is introduced, which allows a certain prediction of buffer hits and misses for tree search and update operations.

## 2. The replacement strategy

A search operation in a B- or B+-tree walks from the root to the leaves [1]. Searching an item X starts with the root node. Depending on the data found in this node, a node from the next deeper level is selected. This continues until a leaf node is reached. So, on each tree level a single node is used. The node needed on level i+1 depends on the result of level i.

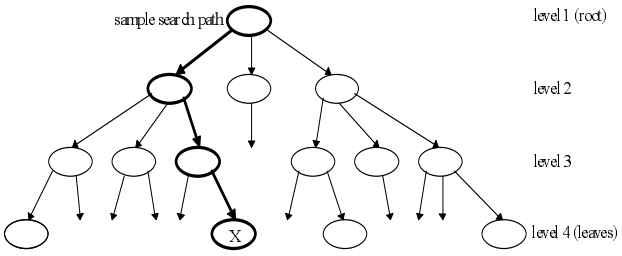


**Figure 1. A sample search path through a tree**

Standard buffer replacement strategies such as LRU, LRD, LFU, CLK, FIFO, etc. [2][3] use the knowledge about the past to estimate future behavior. By analyzing the past, e.g. how often each node in the buffer was used until now, they try to find a node to replace, which has the least likelihood to be reused in the future. For real-time database applications, these strategies have been enhanced to respect transaction priorities [4][5][6]. But for all these strategies a buffer hit or miss can only be predicted for a specific known node.

So if one of these strategies is used for B- or B+-trees, it is impossible to predict the buffer hits or misses for a search operation, because the nodes needed during the walk through the tree are unknown at the beginning of the search operation.

To avoid this, the replacement strategy presented here uses other principles, which allow the prediction of buffer hits and misses for a tree search, even if the value of the item searched itself is not known.

This strategy called ***LC*** (Level Control) uses the structural information known about the trees and the tree search to determine a node to be replaced. The basic idea is to use the level of a node as replacement criteria. The replacement priority $R_n$ of a node n is calculated by the node's distance from the root level, which is level 1.

$$R_n = \text{level}(n), \quad \text{where: level(root)} = 1 \qquad (1)$$

The node with the highest replacement priority will be replaced.

This strategy has two advantages: Firstly, it should be efficient. Because only a single node is used on each tree-level during a search operation, the usage probability $P_n$ of a node depends mainly on the number of nodes sharing its level. The usage probability of the root node is 1. In general, $P_n$ can be calculated as:

$$P_n = 1/\text{nodes}(\text{level}(n)), \qquad (2)$$

where: nodes(level(n)) = number of nodes on the level of node n

Supposing a B- or $B^+$- tree with an average of f successors for each node, $P_n$ can be estimated as a function of a node's level:

$$P_n \approx 1/f^{\,\text{level}(n)-1} \qquad (3)$$

It is obvious, that the usage probability of a node is as higher as closer this node is to the root level. So the proposed replacement strategy favors nodes with high usage probabilities.

The second advantage using this strategy is the predictability of buffer hits and misses. As shown in figure 2, the LC-strategy causes the following buffer organization: The first L levels (1 .. L) of a tree are completely stored in the buffer. Level L+1 is stored partially. The remaining levels (L+2 .. $L_L$) are not stored. The value of L depends on the available buffer size for that tree and on the size of the tree itself. So it is very simple to calculate the number of buffer hits and misses during a tree search operation. In worst case, we have:

$$\mathbf{Hits_w = L} \qquad \mathbf{(4)}$$

$$\mathbf{Misses_w = L_L - L} \qquad \mathbf{(5)}$$

In best case, we have:

$$\mathbf{Hits_b = L + 1} \qquad \mathbf{(6)}$$

$$\mathbf{Misses_b = L_L - (L + 1)} \qquad \mathbf{(7)}$$

This is a good prediction, because the difference between best and worst case is only 1. Based on (4) .. (7), resulting best and worst buffer hit rates during a tree search operation can be calculated as:

$$\mathbf{Hitrate_w = L / L_L} \qquad \mathbf{(8)}$$

$$\mathbf{Hitrate_b = (L + 1) / L_L} \qquad \mathbf{(9)}$$

So if only L and $L_L$ are known (which is not difficult in a concrete buffer management implementation based on LC), the number of buffer hits and misses in worst and best case can be calculated independent of a specific search item's value.

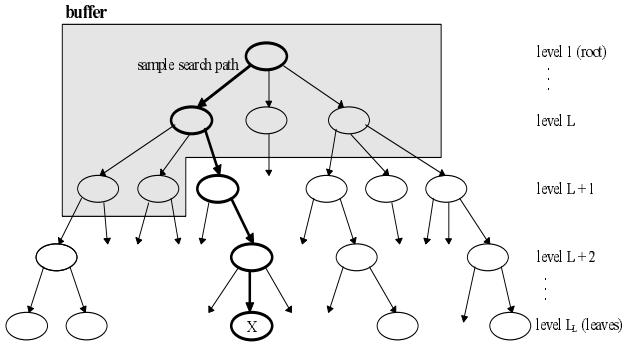


**Figure 2. A tree stored in buffer using the LC-strategy**

If the number of stored nodes $s_t$ of a B- or $B^+$-tree with m to 2 m successors per node is known, the worst and best case for the value of L can be calculated as well:

$$\lfloor \log_{2m+1}(2\,m\,s_t + 1) \rfloor \le L \le \lfloor \log_{m+1}((s_t - 1)\,m/2 + 1) + 1 \rfloor \qquad (10)$$

($\lfloor\ \rfloor$ = floor function)

$$\Rightarrow \quad \mathbf{Hitrate_w = \lfloor \log_{2m+1}(2\,m\,s_t + 1) \rfloor / L_L} \qquad \mathbf{(11)}$$

$$\mathbf{Hitrate_b = (\lfloor \log_{m+1}((s_t - 1)\,m/2 + 1) + 1 \rfloor + 1) / L_L} \qquad \mathbf{(12)}$$

Of course, hit rates calculated by (8) and (9) are more precise than those by (11) and (12), because equations (8) and (9) use the actual value of L, equations (11) and (12) use worst and best case values of L.

Additionally, if the total number of nodes and the number of nodes stored in buffer for level L + 1 are known, the probability of the best case hit rate can be calculated:

$$P_b = nodes_{buffered}(L + 1) / nodes(L + 1) \qquad (13)$$

If the total number of nodes on level L+1 is unknown, an upper limit can be given:

$$nodes(L + 1) \leq 2\ m^{L+1} \qquad (14)$$

$$\Rightarrow \qquad P_b \geq nodes_{buffered}(L + 1) / 2\ m^{L+1} \qquad (15)$$

So far, we have analyzed search operations in B- and B$^+$-trees using LC as buffer replacement strategy. Finally let's have a look at insert and delete operations. Each insert and delete operation is preceded by a search operation. In best case, only a single value in one of the nodes accessed has to be removed or inserted. No further access to any other node is necessary. This means, in best case the number of misses and the hit rate is equal to the one for search operations (7):

$$\mathbf{InsDelMisses_b = L_L - (L + 1)} \qquad \mathbf{(16)}$$

In worst case, a second node has to be accessed on each tree level due to splitting, melting or compensating tree nodes. This means, in worst case the number of misses for insert and delete operations doubles (5):

$$\mathbf{InsDelMisses_w = 2\ (L_L - L)} \qquad \mathbf{(17)}$$

## 3. Simulation results and comparison to other strategies

To confirm the results and to compare the LC strategy to other strategies, several simulations have been made. Figures 3 a, b and c show some of the measured hit rates for search operations in different trees compared to the calculated worst and best case hit rates based on equations (11) and (12). As expected, the calculated worst case hit rate is the lower limit in all cases. For slender trees (low values of m), the hit rate gets closer to the best case hit rate. This behavior can be explained by equation (15). If the value of m decreases, the probability $P_b$ for best case hit rates increases. So as expected, equations (4) to (7) produce a good and small prediction interval for buffer hits and misses.

Figures 4 a, b and c show some of the simulation results, that compare the LC strategy to other common replacement strategies. All hit rates are measured under identical conditions. Furthermore, as a measure of quality, an 'optimal strategy' [7] is used, which is unrealizable outside a simulation. This strategy uses

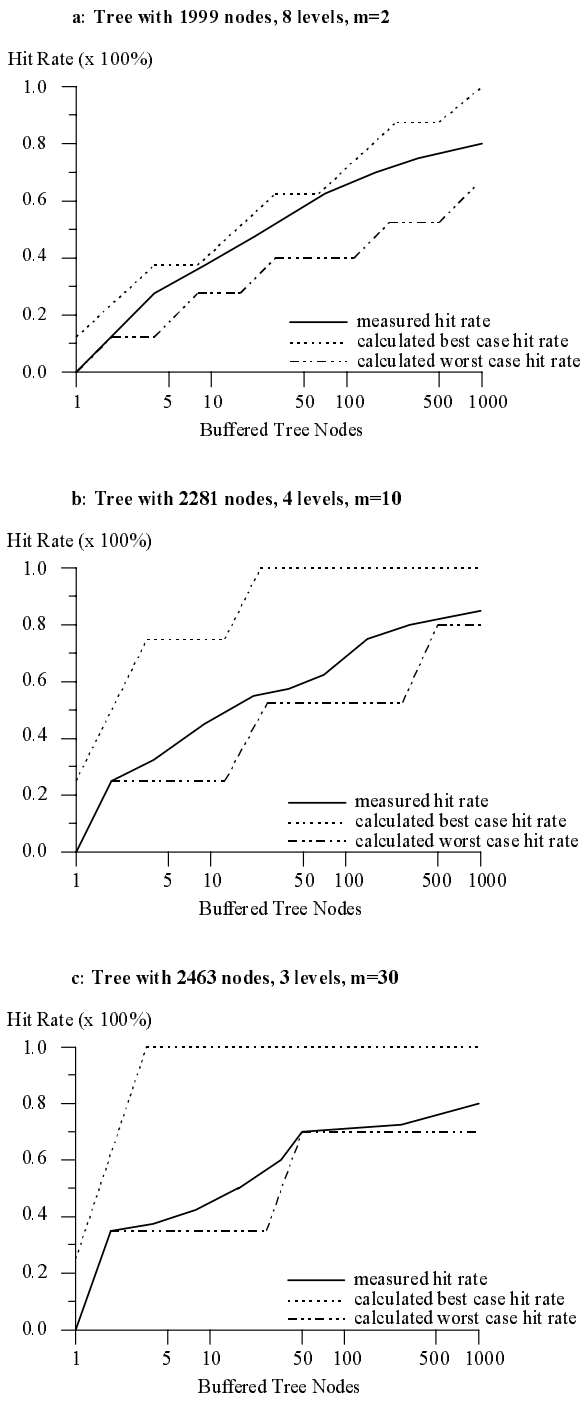


**Figure 3 a, b, c. Calculated and measured hit rates of the LC-strategy**

knowledge about the future to determine the node, that will be unused for the longest time in future. So the 'optimal strategy' can be considered as an upper limit to all realizable strategies.

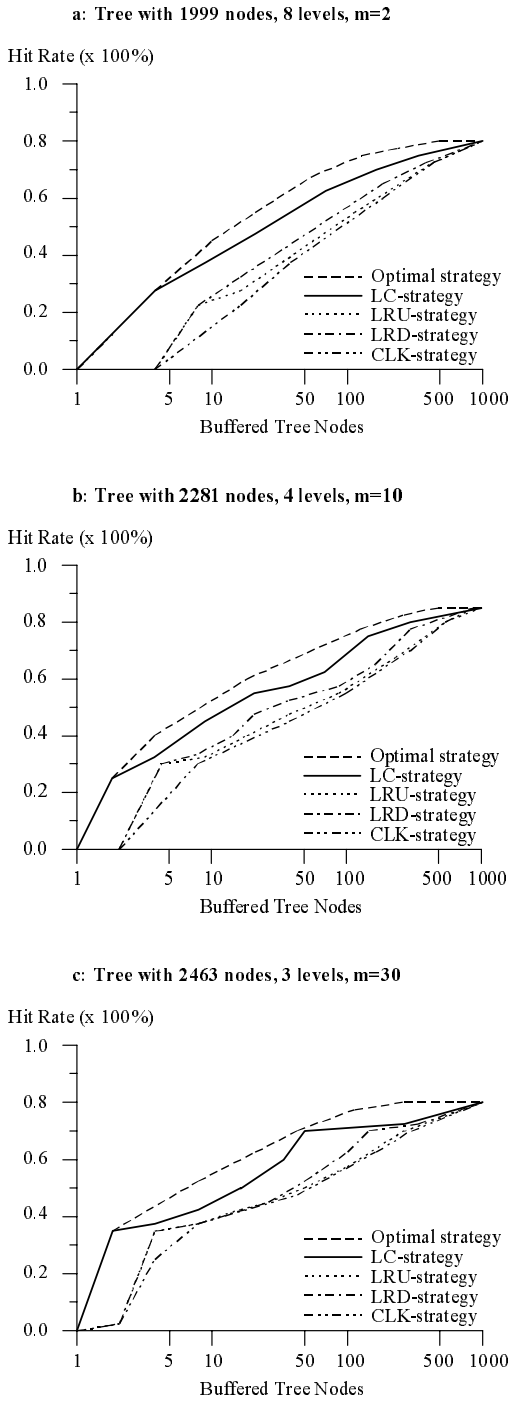


**Figure 4 a, b, c. Hit rate comparison of different strategies**

The comparison shows, that the LC strategy is a very efficient strategy. In nearly all simulations made, the LC strategy got the best results of the compared realizable strategies. Sometimes it even comes close to the 'optimal strategy'. Especially for a small number of nodes buffered, the LC strategy has significant hit rate advantages. This is because upper levels of a tree contain a small number of nodes compared to the lower levels, but each level is used exactly once during a search operation. So the LC strategy can be considered efficient in hit rates and in the use of buffer capacities.

## 4. Applications

The LC strategy can be applied to realize an efficient buffer management for B- or $B^+$-trees in a real-time database system kernel. If this management software maintains the values of L and $L_L$, the number of buffer misses for each tree access can be predicted. A first implementation of LC was made for the soft real-time database kernel Merlin [8]. A second implementation is planned for a scaleable firm and hard real-time database kernel, which is in the conception phase at the moment.

In a real-time database application, some enhancements to the basic strategy can be made without affecting predictability: First it is very likely, that there is more than one node with the same highest replacement priority. To decide between these nodes, a secondary strategy is needed. A good secondary strategy, e.g. LRU, can increase the value of $P_b$.

Furthermore, in a database application more than one tree is stored in buffer at the same time. Several trees must share the buffer. Some of those trees may be used more often than others. To respect this, trees can be given a usage-dependend weight to increases the value of L for often used trees.

Transaction priorities and deadlines may affect the tree weights and the secondary strategy as well.

## 5. Conclusions

The presented LC strategy is an efficient and predictable buffer replacement strategy for B- and $B^+$-trees. Buffer hits and misses during search operations can be predicted in small intervals of constant size 1. During update operations, the prediction interval is of larger size ($L_L$ - L + 1), but a worst case value can be guarantied as well. So LC seems to be a good strategy for access-path buffer management based on B- or $B^+$-trees in real-time database system kernels.

With some modifications, it should be furthermore possible, to adapt the LC strategy to other types of trees as well.

## 6. References


[1] R. Bayer and E. M. McCreight, "Organisation and Maintenance of Large Ordered Indexes", *Acta Informatica*, Volume 1, Number 3, 1972, pp. 173-189

[2] W. Effelsberg and T. Härder, "Principles of Database Buffer Management", *ACM Transactions on Database Systems*, Volume 9, Number 4, 1984, pp. 560-565

[3] P.C. Lockemann and J.W. Schmidt, *Datenbank-Handbuch*, Springer Verlag, Berlin, 1987

[4] J. Huang and J.A. Stankovic, "Buffer Management in Real-Time Databases", *COINS Technical Report* 90-65, University of Massachusetts, 1990

[5] M.J. Carey, R. Jauhari and M. Livny, "Priority in DBMS Resource Scheduling", *15th. Int. Conf. on Very Large Databases*, Amsterdam, 1989, pp. 397-410

[6] H. Pang, M.J. Carey and M. Livny, "Managing Memory for Real-Time Queries", *ACM Sigmod Int. Conf. on the Management of Data*, Minneapolis, 1994, pp. 221-232

[7] L.A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers", *IBM Systems Journal*, Volume 5, Number 2, 1960, pp. 78-101

[8] H. Vogelsang, U. Brinkschulte and M. Siormanolakis, "Archiving System States by Persistent Objects", *ECBS'96 International IEEE Symposium and Workshop on Engineering of Computer Based Systems*, Friedrichshafen, Germany, 1996, pp. 292-297